CSCI 136
Data Structures & Advanced Programming

Lecture 11

Spring 2020

Bill J & Dan

Administrative Details

- Lab 4
  - Another Partner Lab
  - New checkstyle rule
  - Less code, more thinking
- Things I feel strongly about:
  - Draw Pictures!
  - Pair program!
- Any announcements?

Last Time

- Abstraction
  - Interfaces = Contract
  - Abstract classes = Contract + implementation
- So far, we've explored (through Vector) the:
  - List Interface
  - AbstractList base class

Today

- Implementing Lists with linked structures
  - Singly Linked Lists
  - Circularly Linked Lists
  - Doubly Linked Lists

- How does each differ from Vector?

The List Interface

```
interface List {
    size()
    isEmpty()
    contains(e)
    get(i)
    set(i, e)
    add(i, e)
    remove(i)
    addFirst(e)
    getLast()
    .
    .
    .
}
```

- It's an interface…therefore it provides no implementation
- Can be used to describe many different types of lists
- `Vector` implements `List`
- Other implementations are possible…

Pros and Cons of Vectors

Pros

- Good general purpose list
- Dynamically resizeable
- Fast access to elements
  - `vec.get(387425)` finds item 387425 in the same number of operations regardless of `vec`'s size

Cons

- Slow updates to front of list (why?)
- Hard to predict time for add (depends on internal array size, which is hidden)
- Potentially wasted space

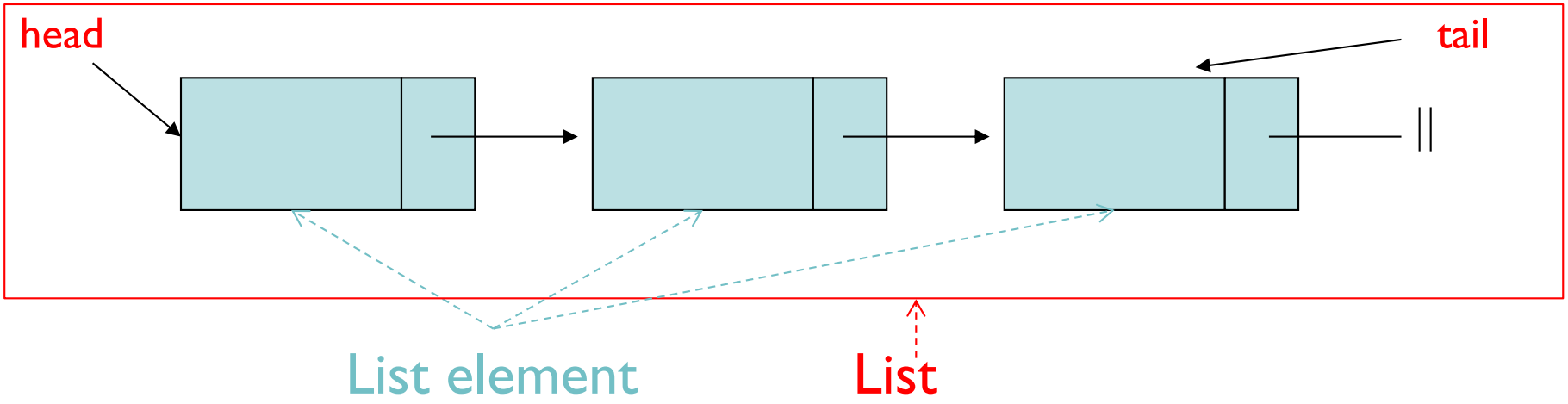
What if we didn't have to copy the array each time we grew `vec`?

List Implementations

- List is a general concept for storing/organizing data
- `Vector` implements the `List` interface
- We'll now explore other `List` implementations
  - `SinglyLinkedList`
  - `CircularlyLinkedList`
  - `DoublyLinkedList`

Linked List Basics

- There are two key aspects of Lists
  - Elements of the list
    - Store data, point to the "next" element
  - The list itself
    - Includes `head` (sometimes `tail`) member variable
- Visualizing lists



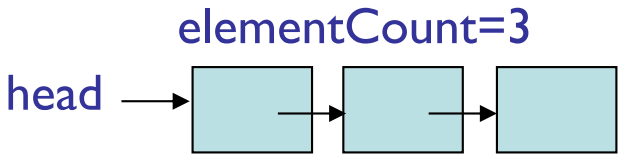
Linked List Basics

- List nodes are recursive data structures
- Each "node" has:
  - A data value
  - A next variable that identifies the next element in the list
  - Can also have "previous" that identifies the previous element ("doubly-linked" lists)
- What methods does the `Node` class need?

SinglyLinkedLists

- How would we implement `SinglyLinkedListNode`?
  - `SinglyLinkedListNode` = SLLN in my notes
  - SLLN = `Node` in the book (in Ch 9)

- How about `SinglyLinkedList`?
  - `SinglyLinkedList` = SLL in my notes



- What would the following look like?
  - `addFirst(E d)`
  - `getFirst()`?
  - `addLast(E d)`? (more interesting)
  - `getLast()`?

More SLL Methods

- How would we implement:
  - `get(int index)`, `set(E d, int index)`
  - `add(E d, int index)`, `remove(int index)`
    - `removeLast()` is just `remove(size() - 1)`
    - `removeFirst()` is just `remove(0)`
- Left as an exercise:
  - `contains(E d)`
  - `clear()`
- Note: `E` is value type (generic)

Get and Set

```
//pre: index < size() – 1, size() > 0
public E get(int index) {
   SLLN finger = head;
   for (int i=0; i<index; i++){
        finger = finger.next();
   }
   return finger.value();
}

//pre: index < size() – 1, size() > 0
public E set(E d, int index) {
   SLLN finger = head;
   for (int i=0; i<index; i++){
        finger = finger.next();
   }
   E old = finger.value();
   finger.setValue(d);
   return old;
}
```

We should add error-checking in our functions. Preconditions aren't enforced by the Java language!

Add

```
public void add(E d, int index) {
   if(index > size()) return null;
   E old;

   if (index==0) { addFirst(d); }

   else if (index==size()) { addLast(d); }

   else {
        SLLN finger = head;
        SLLN previous = null;
        for (int i=0; i<index; i++) {
            previous = finger;
            finger = finger.next();
        }
        SLLN elem = new SLLN(d, finger);
        previous.setNext(elem); // new “ith” item added after i-1
        count++;
   }
}
```

Remove

```
public E remove(int index) {
   if(index >= size()) return null;

   E old;

   if (index==0) {              // Special case: remove from head
        old = head.value();
        head = head.next();
        count--;
        return old;
   }

   else {
        SLLN finger = head;
        for (int i=0; i<index - 1; i++) { //stop one before index
                finger = finger.next();
        }
        old = finger.next.value();
        finger.setNext(finger.next().next());
        count--;
        return old;
   }
}
```

Linked Lists Summary

- Recursive data structures used for storing data
- More control over space use than Vectors
- Easy to add objects to front of list
- Components of SLL (`SinglyLinkedList`)
  - `head`, `elementCount`
- Components of SLLN (`Node`):
  - `next`, `value`

Vectors vs. SLL

- Compare performance of:
  - `size()`
  - `addLast()`, `removeLast()`, `getLast()`
  - `addFirst()`, `removeFirst()`, `getFirst()`
  - `get(int index)`, `set(E d, int index)`
  - `remove(int index)`
  - `contains(E d)`
  - `remove(E d)`

SLL Summary

- SLLs provide methods for efficiently modifying front of list
  - Modifying tail/middle of list is not quite as efficient
- SLL runtimes are consistent
  - No hidden costs like Vector.ensureCapacity()
  - Avg and worst case are always the same
- Space usage
  - No empty slots like vectors
  - But keep extra reference for each value
    - overhead proportial to list length
      - (but this is constant and predictable)

DoublyLinkedLists

- Keep reference/links in **both** directions
  - previous and next
- DoublyLinkedListNode instance variables
  - DLLN next, DLLN prev, E value
- Space overhead is proportional to number of elements
- ALL operations on tail (including removeLast) are fast!
- Additional complexity in each list operation
  - Example: add(E d, int index)
  - Four cases to consider now: empty list, add to front, add to tail, add in middle

```
public class DoublyLinkedNode<E>
{
	protected E data;
	protected DoublyLinkedNode<E> nextElement;
	protected DoublyLinkedNode<E> previousElement;

// Constructor inserts new node between existing nodes
public DoublyLinkedNode(E v,
			DoublyLinkedNode<E> next,
			DoublyLinkedNode<E> previous)
{
	data = v;
	nextElement = next;
	if (nextElement != null)
		nextElement.previousElement = this;
	previousElement = previous;
	if (previousElement != null)
		previousElement.nextElement = this;
}
```

CircularlyLinkedLists

- Use *next* reference of last element to reference head of list
- Replace ***head*** reference with ***tail*** reference
- Access head of list via *tail.next*
- ALL operations on head are fast!
- addLast() is still fast
- Only modest additional complexity in implementation
- Can "cyclically reorder" list by changing *tail* node
- Question: What's a circularly linked list of size 1?

Food for Thought: SLL Improvements to Tail Ops

- In addition to Node head and int elementCount, add Node tail reference to SLL
- Result
  - addLast and getLast are fast
  - removeLast is not improved
    - We need to know element before tail so we can reset tail pointer
- Side effects
  - We now have three cases to consider in method implementations: empty list, head == tail, head != tail
  - Think about addFirst(E d) and addLast(E d)